A PARALLEL BRANCH AND BOUND ALGORITHM FOR TEST GENERATION †

Srinivas Patil and Prith Banerjee

Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

ABSTRACT

For circuits of VLSI complexity, test generation time can be prohibitive. Most of the time is consumed by hard-to-detect (HTD) faults which might remain undetected even after a large number of backtracks. We identify the problems inherent in a uniprocessor implementation of a test generation algorithm and propose a parallel test generation algorithm which tries to achieve a high fault coverage for HTD faults in a reasonable amount of time. A dynamic search space allocation strategy is used which ensures that the search spaces allocated to different processors are disjoint. The parallel test generation algorithm has been implemented on an Intel iPSC/2 hypercube. Results are presented using the ISCAS combinational benchmark circuits which conclusively prove that parallel processing of HTD faults does indeed result in high fault coverage which is otherwise not achievable by a uniprocessor algorithm in limited CPU time. The parallel algorithm exhibits superlinear speedups in some cases due to search anomalies.

1. Introduction

The problem of test generation has been shown to be NP-complete[1] even for combinational circuits. Algorithms for test generation normally use heuristics to guide the search for a test. One of the most widely used algorithms is the D-algorithm [2]. But D-algorithm has been shown to be inefficient in the case of error correction and translation (ECAT) circuits which are characterized by a large number of Exclusive-Or gates. An algorithm called PODEM [3] was proposed to remedy this problem. This algorithm views the test generation process as an implicit enumeration of an n-dimensional boolean search space (n is the number of primary inputs of the logic circuit). A variant of PODEM called the FAN algorithm [4] was proposed which uses much better heuristics to guide the search. More recently, some simulation based methods have been proposed which use a cost function to guide the search for a test vector [5]. For circuits of VLSI complexity, the test generation for a whole chip may take days and even weeks to complete. Improved heuristics lead to only a marginal improvement in run time for most of these algorithms. The only way to speed up the task of test generation by orders of magnitude is to use multiprocessors [6, 7]. With general purpose multiprocessors becoming more commonplace, this seems to be a very attractive alternative.

Parallel search techniques have been applied to different problems in the area of Artificial Intelligence e.g., the 15-puzzle problem [8]. In this paper we present a parallel branch and bound formulation of test generation. We then present our results based on implementation of PODEM using this strategy on a 16 processor Intel iPSC/2 hypercube using the ISCAS benchmark circuits. We show through experiments the resulting improvement in fault coverage as well as time efficiency due to the proposed technique. Due to search anomalies, the parallel algorithm exhibits superlinear speedups in some cases. Throughout our discussion we shall assume that the reader is familiar with test generation algorithms, specifically PODEM [3]. Combinational gate-level logic circuits are assumed.

2. Exploiting Parallelism in Test Generation

There are various ways in which we can parallelize test generation. *Fault parallelism* refers to the evaluation of the given fault set in parallel. The fault set is divided equally among n processors. In an integrated test generation/ fault simulation environment, test generation and fault simulation are interleaved to reduce the overall run time and to produce a compact test set. In a parallel environment, each processor fault simulates test vectors derived during its own test generation phase and by other processors. Communication among processors for fault simulation can be reduced or altogether eliminated by proper partitioning of the fault set. The main advantage of fault parallelism is that the communication overhead is very low and it is possible to achieve linear speedups when the number of processors is very small compared to the number of faults. But the main disadvantage is that faults which are hard to detect for the uniprocessor algorithm (i.e., faults which require a large number of backtracks on the uniprocessor) remain undetected.

As mentioned earlier, almost all test generation algorithms use some heuristic or the other to guide test generation. Experiments reported have suggested that there is no clear cut advantage of using one heuristic over the other [9]. We can exploit *heuristic parallelism* by allowing each processor to use a different heuristic to guide the search for the same fault. Superlinear speedups have been predicted in some cases [10]. But the main disadvantage of this method is that the parallelism is limited by the number of heuristics available for search (at most 5-6). Also, no improvement is possible if a fault remains undetectable for all the heuristics.

One major component of run time in test generation is forward implication. Every primary input assignment is followed by forward implication which takes $O(n)$ time in the worst case (n is the number of lines in the logic circuit). This makes backtracks expensive and the test generation process has to be aborted when a test cannot be generated after a certain backtrack limit. We can

† This research was supported in part by the Joint Services Research Program under Contract N00014-84-C-0149, in part by the Semiconductor Research Corporation under Contract SRC 87 DP-109 and in part by the National Science Foundation under grant NSF CCR 87-05240.

exploit *simulation parallelism* by using parallel logic simulation methods to speed up forward implication. The circuit can be partitioned across processors so that each processor simulates a part of the circuit. The partitioning strategy should minimize communication and maximize concurrency.

Another way to view the test generation process is as a series of subtasks. This leads to an AND/OR type parallelism as described in [11]. *AND parallelism* arises in algorithms using a divide and conquer type approach where a task is divided into a series of subtasks, all of which have to be completed. Thus these subtasks can be completed in parallel if they do not access or modify any common variables. If they have common variables, then execution of the AND-parallel subtasks becomes complicated due to variable binding conflicts. The overheads due to variable binding conflicts may essentially reduce the AND-parallel tasks to sequential execution due to consistency checking and synchronization overheads.

OR parallelism refers to the concurrent evaluation of choice points. In the case of test generation, this can be achieved by searching different portions of the search space concurrently. Each processor can be allocated a portion of the search space, either statically or dynamically and the search for the test vector can proceed concurrently. A brute force method of dividing the search space might entail useless search in non-solution areas by some processors. A heuristics based method aimed at increasing the probability of searching in a solution area would be more desirable but the search spaces may not be disjoint in such a situation. The advantage of OR parallelism is that communication overhead is very low and a large number of processors can be utilized efficiently. Also, the backtracks per processor can be reduced. Superlinear speedups might be possible due to search anomalies [12]. The proposed parallel branch and bound method can be considered a high-level form of OR parallelism.

3. Test Generation Using Parallel Branch and Bound

PODEM treats test generation as a branch and bound search on the n-dimensional Boolean input space of a logic circuit. Even though a test may exist for a fault, the search may have to be terminated after a predecided backtrack or time limit to save on CPU time. An algorithm which follows the shortest path to a test vector during the search takes the shortest time. One way in which test generation algorithms try to follow paths of shorter lengths is by using testability measures[13, 14] to guide the search. But the problem of calculating exact testability measures for a logic circuit with reconvergent fanout has an exponential worst case time complexity [15]. Also, it might be necessary to recalculate testability measures during the test generation process as the circuit state changes. It has been experimentally shown that this does reduce the number of backtracks but the overheads due to recomputing the testability measures every time a primary input assignment is made overshadow any gains due to fewer number of backtracks [16]. Since it is not practical to calculate exact testability measures and recomputing testability measures is expensive, most test generation algorithms use static and approximate testability measures. The heuristic guidance provided by approximate static testability measures may not lead to a solution along a shorter path even though one might exist. As suggested in [9, 10] the alternative would be to use multiple testability measures. On a uniprocessor one may switch between different heuristics or on a multiprocessor implementation we may be able to pursue solutions using different heuristics in parallel. The main drawback of such a method is that there is no way to ensure that the search spaces are disjoint. Wrong paths taken by one heuristic might be repeated by another heuristic and the fault might remain undetected even if a test exists. As pointed out earlier, on a multiprocessor, the parallelism is limited by the number of heuristics available for search and two processors running on two different heuristics might end up doing redundant work because their search spaces may not be disjoint. The multiprocessor implementation offers no improvement if a set of faults remains undetected by all heuristics. The method we propose offers an improvement over the uniprocessor algorithm both in terms of time and fault coverage irrespective of the heuristic used and it is possible to utilize a large number of processors efficiently.

The decision tree in PODEM is a binary tree and is maintained as a stack. Each stack entry has three fields - a *val* field storing the current assignment to a primary input, an *input* field identifying the primary input and a *tried* field indicating whether the alternative value at the primary input has been tried. The algorithm proceeds in a depth-first manner to implicitly enumerate the search space for a test vector. If a conflict occurs during this process, the algorithm backtracks to a previous choice point to try another alternative. It has been shown experimentally in [9] that it is very unlikely that a fault which has not been detected by heuristic with a lower backtrack limit will be detected by increasing the backtrack limit. There is a very fundamental reason to this observation. When a conflict occurs, the assignment which gave rise to the conflict need not necessarily be the assignment made at the previous choice point. A blind backtrack to the previous choice point may lead to more fruitless backtracks. So, the algorithm would perform much better if one were to backtrack to the point which gave rise to the conflict. This is easier said than done. It is very difficult to determine which previous assignment gave rise to the conflict. The proposed parallel implementation tries to divide the search space so that intermediate choice points are skipped by one processor while at the same time the skipped choice points are evaluated by some other processor. This leads to a much better fault coverage figure for hard-to-detect (HTD) faults compared to the uniprocessor implementation. Also, the search space is divided so that all processors are searching disjoint subspaces at all times. Since the size of the search space is exponential in the number of primary inputs, many processors can be utilized efficiently.

Figure 1 shows how a decision tree belonging to processor i would be split between processor i and another processor j. The choice points are alternately split between processor i and processor j so that each processor has roughly equal number of choice points. The exact mechanism by which these search spaces are allocated is described in the next section.

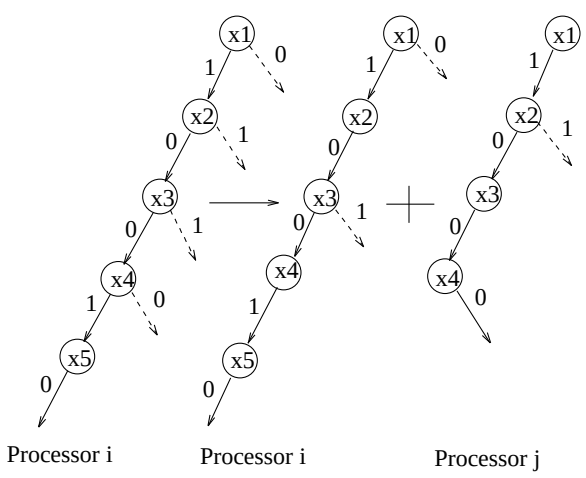

Figure 1. Splitting search spaces

4. Implementation Details

The PODEM-based test generator using parallel branch and bound was implemented on a 16 node Intel iPSC/2 hypercube. A dynamic search-space allocation strategy was used to keep the load balanced at all times. The nodes in an n-dimensional hypercube are numbered from 0 to $(2^n - 1)$. One of the nodes (node 0) is used for scheduling (known as a *scheduler*) while nodes 1 to $(2^n - 1)$ carry out test generation using parallel branch and bound (known as *processing* nodes). The scheduler can communicate with all the processing nodes and the host while a processing node is allowed to communicate with only the scheduler. Initially, the circuit description and information about the heuristic to be used is broadcast by the host to all the nodes. For each fault, the host broadcasts the gate-under-test and the type of fault (s-a-1 or s-a-0) and waits for the result from the scheduler. The processing nodes communicate with the scheduler to get more work or to inform the scheduler of special conditions. The types of messages passed between the scheduler and the processing nodes are as follows:

NOTEST: Sent by a processing node to the scheduler to indicate that a test is not possible for a particular fault with the given search subspace. Also sent by the scheduler to the host to indicate that no test exists for the given fault.

ABORT: Sent by a processing node to a scheduler if a processing node aborts test generation after hitting the backtrack limit. Also sent by the scheduler to the host to indicate that test generation for the given fault had to be aborted because the backtrack limit was exceeded.

TESTED: Sent by a processing node to a scheduler if a processing node generates a test for the given fault. Also sent by the scheduler to the host if a test is generated for the given fault.

WRK_REQ: Sent by the scheduler to a processing node to request for more work.

DONE: An abort message sent by the scheduler to all processing nodes to stop test generation for a particular fault. Also sent by the processing nodes to the scheduler to acknowledge a DONE message.

WORK: A message sent by a processing node to a scheduler in response to a WRK_REQ message. Also used by the scheduler to send work to a processor which requested for work through a NOTEST message.

The function of the scheduler is to coordinate the activities of the processing nodes. It receives messages from the processing nodes and carries out actions based on the type of message. The host broadcasts gate-under-test and the fault type (s-a-1 or s-a-0) to each node. The scheduler maintains a list of processing nodes which are busy generating a test for the current fault. This list is maintained in the form of a queue and is called a *donor queue*. Initially, node 1 starts generating a test for the fault assuming it has the whole search space. At the beginning of test generation for a particular fault, the donor queue is initialized to include node 1. The remaining processing nodes do not send an explicit message for requesting work but send a NOTEST message indicating that the search space is empty and that they require more work. The actions executed by the scheduler on receiving a NOTEST message is given by the following procedure:

procedure *allocate-work*:

(1) If there is an entry for the node that sent the NOTEST message in the donor queue then delete it from the donor queue.

(2) Select a donor from the top of the donor queue. If the donor queue is empty then enter the *wrap-up phase*. Otherwise, put the donor at the end of the queue. This is to make sure the same node is not asked repeatedly for work and the search spaces being processed by each node are uniformly distributed over the initial search space.

(3) Send a WRK_REQ message to the donor. Wait for work to arrive from the donor. Send the work to the node that sent a NOTEST message. If the work received is non-null, then put the node that sent the NOTEST message at the end of the donor queue. If the work received is null, then delete the donor from the donor queue.

end *allocate-work*.

On receiving an ABORT message, the scheduler deletes the node which sent the ABORT message from the donor queue. If the donor queue becomes empty after this action, then the scheduler enters the wrap-up phase. On receiving a TESTED message from a node, the scheduler enters the wrap-up phase unconditionally.

The actions executed by the scheduler in the wrap-up phase are as follows:

procedure *wrap-up*:

(1) Broadcast a DONE message to all the processing nodes. Then the scheduler waits for a DONE message from each of the nodes. This is to make sure all the messages from the processing nodes for the current fault have been flushed.

(2) If any of the processing nodes sent a TESTED message, then send the test vector to the host. Otherwise, if any of the processing nodes sent an ABORT message, then send an ABORT message to the host else send a NOTEST message.

end *wrap-up*.

The processing nodes can either be in the *waiting phase* or *PODEM phase* after receiving the gate-under-test and the fault type from the host. In the waiting phase, a node waits for work from the scheduler. In the PODEM phase a node executes PODEM on its own search phase to generate a test for the given fault. After receiving the current fault from the host, node 1 enters the PODEM phase directly assuming it has the whole search space. Other processing nodes send a NOTEST message to the scheduler indicating they need work.

The actions performed by the processing nodes in PODEM phase are as follows:

procedure *podem*:

(1) In the main PODEM loop periodically check for messages from the scheduler.

(2) If a test is generated for the given fault, send a TESTED message along with the test vector to the scheduler. Enter the *wind-up phase*.

(3) If backtrack limit is exceeded, send an ABORT message to the scheduler and enter the wind-up phase.

(4) If the search space is exhausted, send a NOTEST message to the scheduler and enter the waiting phase.

(5) If a WRK_REQ message is received, then split the stack as shown in Figure 1 and send it to the scheduler.

(6) If a DONE message is received, enter the wind-up phase.

end *podem*.

The actions performed by the processing nodes in the waiting phase are as follows:

procedure *wait-for-work*:

(1) If a WRK_REQ message is received, send a WORK message of size zero to the scheduler to indicate no work is

available.

(2) If a WORK message is received and the size of the work is non-zero, then enter the PODEM phase. The stack in the PODEM phase is initialized to the stack received as a part of the WORK message. The processing node then performs a forward implication using the values in the received stack to initialize the logic values in the circuit properly. It then computes the D-frontier and enters the PODEM phase. If the size of the WORK message is zero, then send a NOTEST message to the scheduler.

(3) If a DONE message is received enter the wind-up phase.

end *wait-for-work.*

In the wind-up phase, the processing nodes send a DONE message to the scheduler to indicate that they are ready to process the next fault. They then wait for the next fault to arrive from the host and they proceed to generate a test for the next fault in the manner indicated above. If the host fault list is empty, the program terminates.

The scheduler is not allowed to execute PODEM to make sure that the scheduling node does not become a bottleneck as the number of processors is increased. The maximum length of messages between the scheduler and the processing nodes is limited to about 200 bytes to reduce the communication latency. Due to search anomalies, superlinear speedups are possible as borne out by results presented in the next section.

5. Results

The parallel algorithm was implemented on a 16 processor Intel iPSC/2 hypercube in C language. The hard-to-detect (HTD) faults were first filtered out using a uniprocessor implementation of PODEM on the host. The host and the nodes have a 4 MIPS Intel 80386 processor. A backtrack limit of 25 was used for the uniprocessor algorithm and all faults dropped by a particular heuristic were classified as HTD faults. The SCOAP [13] testability measure was used for guidance. It has been shown in [9] that SCOAP performs better than other heuristics on the average. The results of the uniprocessor implementation on the ISCAS benchmark circuits[17] using SCOAP are shown in Table I. The average time per fault was calculated by dividing the time taken by the algorithm for all the faults (including dropped or HTD faults) by the total number of faults. The fault coverage figure is not the fault coverage in the fault simulation sense since no fault simulation was used here. The fault coverage is rather an efficiency figure for the test generation algorithm and is equal to the sum of the faults for which tests were generated and faults which were found redundant divided by the total number of faults.

The faults dropped by SCOAP heuristic were used as HTD faults for the next part of our experiments. Circuit c880 was removed from further consideration since SCOAP was able to generate tests for 100% of the faults. Table II shows the results for HTD faults using parallel branch and bound on 4, 8 and 16 processors. For a fair comparison with the uniprocessor results, the total backtrack limit over all processors was kept the same as the backtrack limit for the uniprocessor implementation. In the parallel implementation, the backtrack limit per processor was kept at 25. For example, since the total backtrack limit over 4 processors is $3\times25=75$ (one processor acts as a scheduler and does not take part in test generation), the uniprocessor backtrack limit was increased to 75. Similarly, for comparison with the 8 processor results, the uniprocessor backtrack limit was 175 and for the 16 processor results, the uniprocessor backtrack limit was 375. The parallel and the uniprocessor algorithms with the scaled up backtrack limits were run on the remaining 9 ISCAS circuits for the HTD faults given in Table II. The purpose of these experiments was three-fold - (1) to show that increasing the backtrack limit for the uniprocessor algorithm does not yield substantial improvements in fault coverage for HTD faults, (2) the parallel implementation yields a better fault coverage in most cases than the uniprocessor algorithm for the same total backtrack limit and (3) the parallel algorithm runs much faster than the uniprocessor algorithm. The numbers in parenthesis in Table II are the corresponding numbers for the uniprocessor algorithm with the same backtrack limit as the parallel implementation. It is quite clear that the parallel implementation performs much better than the uniprocessor version in most cases, both with respect to the HTD fault coverage and the time per HTD fault. For one particular circuit (c6288), the speedup was superlinear due to search anomalies as shown by the speedup curves in Figure 2. The message passing overheads are low since messages are passed only to get work and to inform the scheduler of completion. The overall coverage figures are obtained by adding the HTD coverage to the corresponding coverage in Tables I. The overall coverage for the parallel implementation ranges between 97 to 99.9% in most cases which would be considered a very high figure for most test generation algorithms.

To demonstrate that the efficiency of the parallel algorithm is independent of the testability measure used, the algorithm was implemented using random selection as a testability measure and results similar to those presented for SCOAP heuristic were obtained. The results are not presented here due to space restrictions. The interested reader is referred to [18]. The splitting technique of Figure 1 could also be used as an effective heuristic on the uniprocessor to increase the fault coverage. A random number generator could be used to split the decision tree and to switch between search spaces in a non-deterministic manner to emulate the behavior of the parallel algorithm. Of course, the parallel algorithm would still do better in terms of the overall run time. Efforts are underway to apply the search space splitting technique to test generation for sequential circuits where it is more likely that a majority of the faults are HTD faults.

6. Conclusions

We identified the problems inherent in the implementation of a uniprocessor algorithm for HTD faults. Finally, we proposed a parallel branch and bound formulation of test generation which overcomes the deficiencies of the uniprocessor algorithm.

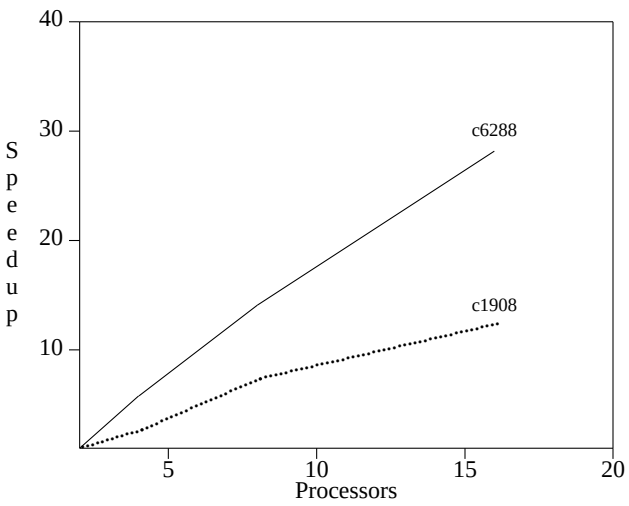

Figure 2. Speedups for HTD faults

Through experiments using the ISCAS combinational benchmark circuits it was shown that the proposed parallel branch and bound technique is able to achieve a greater fault coverage than the uniprocessor algorithm at a fraction of the time consumed by the uniprocessor implementation. In some cases, the parallel implementation showed a superlinear speedup. Thus, parallel processing of HTD faults offers a very cost effective way to achieve high fault coverage. The search space splitting technique could be used as a very effective heuristic for the uniprocessor algorithm to increase the fault coverage. In our current implementation, the central scheduler coordinates the activities of the processing nodes. For larger number of processors, the scheduler could become a bottleneck in the performance of the parallel algorithm. We are investigating techniques for distributed scheduling in a larger system. Our final goal is to build a very efficient parallel test generation system incorporating redundancy identification, parallel fault simulation and parallel test generation.

REFERENCES

[1] H. Fujiwara and S. Toida, ‘‘The complexity of fault detection problems for combinational logic circuits,’’ *IEEE Trans. Computers*, vol. C-31, pp. 555-560, June 1982.

[2] J. P. Roth, ‘‘Diagnosis of automata failures: A calculus and a method,’’ *IBM J. Research Development*, vol. 10, pp. 278-291, July 1966.

[3] P. Goel, ‘‘An implicit enumeration algorithm to generate tests for combinational logic circuits,’’ *IEEE Trans. Computers*, vol. C-30, pp. 215-222, March 1981.

[4] H. Fujiwara and T. Shimono, ‘‘On the acceleration of test generation algorithms,’’ *IEEE Trans. Computers*, vol. C-32, pp. 1137-1144, December 1983.

[5] K. T. Cheng and V. D. Agrawal, ‘‘A simulation-based directed search method for test generation,’’ *Proc. Int. Conf. Computer Design*, pp. 48-51, October 1987.

[6] A. Motohara, K. Nishimura, H. Fujiwara, and I. Shirakawa, ‘‘A parallel scheme for test-pattern generation,’’ *Proc. Int. Conf. Computer-Aided Design*, pp. 156-159, November 1986.

[7] G. A. Kramer, ‘‘Employing massive parallelism in digital ATPG algorithms,’’ *Proc. Int. Test Conf.*, pp. 108-114, 1983.

[8] V. N. Rao and V. Kumar, ‘‘Parallel depth first search, Part I: Implementation,’’ *Int. J. Parallel Programming*, vol. 16, 1987.

[9] S. J. Chandra and J. H. Patel, ‘‘Experimental evaluation of testability measures for test generation,’’ *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 93-98, January 1989.

[10] S. J. Chandra and J. H. Patel, ‘‘Test generation in a parallel processing environment,’’ *Proc. IEEE Int. Conf. Computer Design*, October 1988.

[11] J. S. Conery, ‘‘The AND/OR process model for parallel interpretation of logic programs,’’ Technical Report 204, University of California - Irvine, June 1983.

[12] B. W. Wah, G. J. Li, and C. F. Yu, ‘‘Multiprocessing of combinatorial search problems,’’ *Computer*, vol. 18, pp. 93-108, June 1985.

[13] L. H. Goldstein and E. L. Thigpen, ‘‘SCOAP: Sandia controllability/observability analysis program,’’ *Proc. 17th Design Automation Conf.*, 1980.

[14] R. G. Bennetts, in *Design of Testable Logic Circuits*. Reading, MA: Addison Wesley Publishing Company, 1984.

[15] S. C. Seth, L. Pan, and V. D. Agrawal, ‘‘PREDICT - Probabilistic estimation of digital circuit testability,’’ *Proc. Int. Symp. Fault Tolerant Computing*, pp. 220-225, June 1985.

[16] A. Ivanov and V. K. Agarwal, ‘‘Testability measures - what do they do for ATPG?,’’ *Proc. Int. Test Conf.*, pp. 129-138, September 1986.

[17] F. Brglez and H. Fujiwara, ‘‘Neutral netlist of ten combinational benchmark circuits and a target translator in FORTRAN,’’ *Special session on ATPG and fault simulation, Proc. IEEE Int. Symp. Circuits and Systems*, June 1985.

[18] S. Patil and P. Banerjee, ‘‘TGEN: An efficient test pattern generation system for sequential circuits,’’ CRHC Technical Report, University of Illinois, April 1990.

Circuit	Faults	Tested	Redundant	Dropped	Coverage	Time/Fault (ms)
c432	524	519	0	5	99.05	40
c499	758	750	0	8	98.95	70
c880	942	942	0	0	100.00	34
c1355	1574	1560	0	14	99.11	150
c1908	1879	1863	6	10	99.47	98
c2670	2747	2518	26	203	92.61	109
c3540	3428	3092	47	289	91.57	290
c5315	5350	5236	44	70	98.69	113
c6288	7744	7688	32	24	99.69	369
c7552	7550	7064	52	434	94.25	258

Table I. Uniprocessor results

Circuit	HTD Faults	Processors	HTD Coverage		Overall Coverage		Time/Fault (ms)		Speedup
c432	5	4	0.00	(0.00)	99.05	(99.05)	229	(582)	2.5
		8	20.00	(0.00)	99.24	(99.05)	227	(1316)	5.8
		16	20.00	(0.00)	99.24	(99.05)	241	(2805)	11.6
c499	8	4	0.00	(0.00)	98.95	(98.95)	283	(722)	2.6
		8	0.00	(0.00)	98.95	(98.95)	295	(1611)	5.5
		16	0.00	(0.00)	98.95	(98.95)	374	(3384)	9.0
c1355	14	4	14.29	(0.00)	99.24	(99.11)	408	(1007)	2.5
		8	14.29	(14.29)	99.24	(99.24)	403	(2075)	5.1
		16	28.57	(14.29)	99.36	(99.24)	556	(4159)	7.5
c1908	10	4	10.00	(0.00)	99.52	(99.47)	634	(1634)	2.6
		8	40.00	(0.00)	99.68	(99.47)	507	(3691)	7.3
		16	40.00	(10.00)	99.68	(99.52)	601	(7395)	12.3
c2670	203	4	23.15	(12.81)	94.32	(93.56)	453	(1241)	2.7
		8	40.89	(25.62)	95.63	(94.50)	448	(2516)	5.6
		16	56.65	(31.53)	96.80	(94.94)	457	(4694)	10.3
c3540	289	4	47.75	(22.84)	95.60	(93.49)	726	(2325)	3.2
		8	66.78	(33.91)	97.20	(94.43)	634	(4544)	7.2
		16	76.82	(38.48)	98.05	(94.81)	640	(8796)	13.7
c5315	70	4	60.00	(58.57)	99.48	(99.46)	440	(1362)	3.1
		8	71.43	(68.57)	99.63	(99.59)	414	(2236)	5.4
		16	80.00	(80.00)	99.74	(99.74)	515	(3544)	6.9
c6288	24	4	91.67	(8.33)	99.97	(99.72)	768	(4454)	5.8
		8	91.67	(12.50)	99.97	(99.73)	723	(10191)	14.1
		16	91.67	(20.83)	99.97	(99.75)	766	(21631)	28.2
c7552	434	4	53.69	(29.49)	97.34	(95.95)	612	(1848)	3.0
		8	73.50	(30.88)	98.48	(96.03)	580	(3455)	6.0
		16	75.12	(35.48)	98.57	(96.29)	862	(6589)	7.6

Table II. Parallel results